Efficient and Scalable Compiler-Directed Energy Optimization for Realtime Applications

Po-Kuan Huang
Electrical and Computer Engineering
University of California, Davis
pohuang@ece.ucdavis.edu

Soheil Ghiasi
Electrical and Computer Engineering
University of California, Davis
soheil@ece.ucdavis.edu

Abstract

We present a compilation technique that targets realtime applications running on embedded processors with combined dynamic voltage scaling (DVS) and adaptive body biasing (ABB) capabilities. Considering the delay and energy penalty of switching between operating modes of the processor, our compiler judiciously inserts mode switch instructions in selected locations of the code and generates executable binary that is guaranteed to meet the deadline constraint. More importantly, our algorithm runs very fast and comes reasonably close to the theoretical limit of energy optimization using DVS+ABB. At 65 nm technology, we improve the energy dissipation of the generated code by an average of 11.4% *under deadline constraints. While our technique's improvement in energy dissipation over conventional DVS is marginal (3%) at 130nm, the average improvement continues to grow to* 4.7%, 8.8% *and* 15.4% *for 90nm, 65nm and 45nm technology nodes, respectively. Compared to a recent ILP-based competitor, we improve the runtime by more than three orders of magnitude, while producing improved results.*

1 Introduction

Microprocessors are one of the major contributors to energy consumption in embedded systems. Consequently, a number of circuit-level techniques such as DVS and ABB have been developed to reduce the energy consumption of the processor.

Quadratic dependence of active power on supply voltage, along with the lower order impact of supply voltage on clock frequency has motivated dynamic supply voltage scaling for processors. In this scheme the operating frequency and supply voltage of processors are throttled at runtime to save energy whenever full performance is not required. This technique was very effective in the old technology nodes, where the share of leakage energy in total energy consumption was negligible. The share of leakage energy, however, increases with the scaling of CMOS technology. Hence, conventional dynamic voltage scaling is less effective with advancement of technology[1].

Adaptive body biasing (ABB) is another well-known CMOS design technique that allows runtime adjustment of transistors threshold voltage. Threshold voltage affects both leakage and delay of the transistors. Hence, its effect can be combined with supply voltage scaling to minimize *total power consumption* for a given frequency [11].

We present a compilation methodology that targets embedded processors with joint DVS and ABB capabilities. We investigate hard realtime systems that have to meet the application deadline and have light-weight OS. Our compiler judiciously inserts mode switch instructions in the code, and subsequently generates code that is optimized for overall energy consumption. The generated code is guaranteed to meet the execution deadline over the input data space.

2 Related Work

Extensive research has been done to minimize dynamic power consumption of a CMOS design. Dynamic voltage scaling is utilized in several fabricated academic and commercial processors. However, with the continuing shrinkage of the device sizes, techniques that only target dynamic power will not be effective [1]. Adaptive body biasing has been utilized to reduce the leakage power consumption [6]. Researchers have also studied the application of run-time ABB techniques [2].

Several research groups have proposed static intraprogram voltage scaling [3, 9]. An analytical study of potential power savings using intraprogram DVS is reported in [3]. The authors also propose an ILP-based approach whose savings come reasonably close to the analytical bounds. Compiler and operating system level optimization are co-

978-3-9810801-2-4/DATE07 © 2007 EDAA

ordinated in [9]. None of these techniques consider leakage power, and the effect of technology scaling on the validity of their results.

3 Processor Model and Operating Modes

Our proposed compilation technique targets a processor with combined DVS and ABB capabilities that can operate at several discrete frequencies. According to [11], each frequency has to be associated with a corresponding pair of supply and body bias voltages that allow operation of the processor at that frequency. The combination of the three parameters, i.e., frequency, supply voltage and body bias, constitute an *operating mode* of the processor. The processor is assumed to be able to switch between operating modes by execution of a specialized instruction, referred to as *mode switch instruction*. Execution of a mode switch instruction initiates the process of setting both supply voltage and body bias of the processors to the target mode implied by the instruction. Note that frequency is a function of supply and body bias voltage, and does not need to be specified separately. Execution of the mode switch instruction, or equivalently switching between modes, incurs delay and energy penalty. Both delay and energy penalty depend on the voltage difference of the two modes involved in switching.

We assume that our target processor can operate at 5 different clock frequencies, from $200MHz$ up to $1GHz$ at $200MHz$ steps. We adopt the process technology and processor parameters from Predictive Technology Models [5] and Intel XScale commercial processors, respectively. We obtain the energy optimal supply and body bias voltages corresponding to each frequency by applying the conclusion in [11]. Table 1 demonstrates the characteristics of the operating modes for our target processor in 90nm.

Operating frequency(MHz)	1000	800	600	400	200
Supply voltage(V)	1.63	1.47	1.29	1.11	0.95
Bias voltage(V)	-0.08	-0.17	-0.25	-0.35	-0.47

Table 1. Processor operating modes at 90nm

4 ILP-based Intraprogram Supply and Bias Voltage Scaling

The ILP-based approach aims to achieve this goal by insertion of static mode switch instructions on all of the control flow edges of the application. To formulate the problem as an ILP instance, profiling and simulation should be carried out to capture the frequency of executing each edge of the application control flow graph (CFG), and the average energy dissipation and delay of application basic blocks in each of the operating modes. To determine the appropriate mode for each edge of the CFG, a set of binary decision variables are assigned to each edge of the application CFG. Subsequently, integer linear constraints are formed to guarantee the 1) assignment of each CFG edge to exactly one mode, 2) execution of the application, considering delay penalty when switching modes, within deadline. The objective function would be another integer linear expression that estimates the total energy consumption including energy penalty of mode switches using integer variables [3][10].

ILP-based technique has two major drawbacks. Firstly, ILP is a well-known NP-Hard problem. Therefore, its runtime is not scalable to large programs. Moreover, it inserts a mode switch instruction before entering each basic block (one mode switch per about 5 instructions on average!). Some modes will be redundant, i.e., they set the processor to the mode that it is already operating at, and can be removed using classic compiler optimization passes. Nevertheless, the performance and energy overhead associated with mode switches partially diminishes the savings. In our previous work, we observed that the ILP solving time for typical applications of about two hundred basic blocks exceeds 30 minutes on an ordinary desktop computer[10]. As expected, the runtime grows very fast with increase of program complexity. For example, it took ILP solver more than 6 hours to solve problem instances associated with typical applications of about five hundred basic blocks.

In order to accelerate the solution time, it is reasonable to employ heuristics to reduce the number of constraints in the ILP instance. This would result in a tradeoff between quality of the solution (energy savings) and solution time that might lead to an acceptable balance of the two. In our study, we filtered out the constraints associated with basic blocks that do not significantly contribute to total energy consumption of the application. For example, eliminating some of the constraints in case of *susan* testbench, allowed us to solve the ILP instance on the order of tens of seconds, while degrading the energy consumption by 12%. However, the gap between the energy consumption of original-ILP and simplified-ILP increases with growth of the application size. Consequently, heuristics applied on top of ILP-based approaches should be viewed as temporary solutions that somewhat push the limitations rather than delivering truly scalable strategies.

5 Efficient and Scalable Mode Switching

5.1 Optimal Scaling Frequency

The basic idea of our method is the following: at each point of the execution, by knowing the maximum (i.e. worst

case) number of cycles required to finish the execution of the application, and the time that is left before violating the deadline, the next operating frequency, F_{next}, would be the slowest possible frequency that guarantees executing the application without violating the deadline:

$$F_{next} = \frac{WCRC + S_d}{T_L} \quad (1)$$

where $WCRC$ denotes the *Worst Case Required Cycles* from that specific point to finish the execution of the application, T_L stands for "Time Left" in seconds, and S_d refers to the delay penalty, in cycles, for switching between two modes. The motivation for this scaling equation is run the processor as slow as possible to meet the deadline for the workload. This equation is proved to be theoretically optimal, if continuous frequencies were available [4]. In practice, however, processors can only run at a number of discrete frequencies and the available frequency immediately larger than F_{next} would be the right choice.

5.2 WCRC Calculation

In order to utilize this equation and obtain the next scaling frequency for each node of the control flow graph, we need to estimate $WCRC$ for each node of the application CFG.

Software timing analysis used to calculate the $WCET$ of the embedded application can also be used to estimate the $WCRC$ of the program. We implemented a $WCRC$ calculation algorithm which is similar to the non-enumeration approach proposed in [12]. We calculate the $WCRC$ for each loop in the program by tracking its heaviest path and calculate the $WCRC$ for entire program by traversing the control flow structure of the program in a bottom-up method. We assume that the worst case execution time (WCET) over input data space, and the input associated with it are known. Note that this assumption is not unreasonable, because guaranteeing the execution time without knowledge of WCET and the associated input is not feasible.

5.3 Check Point Insertion

After determining $WCRC$ values for all of CFG edges, we instrument the code to access the time elapsed and the number of cycles executed so far from the operating system. We also need to consider the penalty required for accessing the operating system. We assume that we consume 100 cycles for accessing the time elapsed and number of cycles executed.

Capturing the number of cycles executed so far, enables the compiler to determine the *Remaining Cycles* at each point. The number of remaining cycles is simply the maximum number of cycles required (or $WCRC$ at the entry) minus the number of cycles executed so far. Note that remaining cycles is a function of the input data, and is not generally equal to $WCRC$.

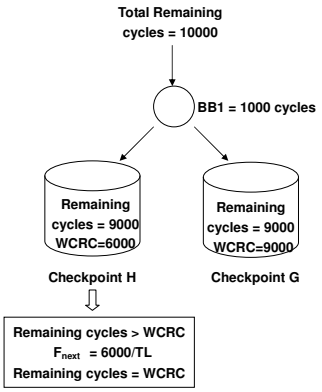

Figure 1. Example function of a check-point

The aforementioned steps are implemented at particular points of execution in regions called *check-points*. Figure 1 illustrates an example check-point and its function.

As shown in Figure 1, we insert checkpoints for edge G and H to update the counters used to track the number of remaining cycles. When the execution goes to check-point G, the number of remaining cycles is 10,000-1000=9,000, because 1000 cycles are spent for executing basic block 1. Checkpoint G finds that the remaining cycles is not greater than $WCRC$ at that checkpoint. It means that it is not safe to lower the frequency at this moment because it would violate the deadline.

On the other hand, if the execution goes to checkpoint H, the number of remaining cycles is still 10,000-1000=9000, however, $WCRC$ for execution of the rest of the program is 6000 cycles. It means that instead of operating under the original frequency, which must be greater than or equal to $9000/T_L$, we can scale down the frequency to $(6000/T_L$ without violating the deadline. After scaling down the frequency, the execution continues and we set the number of remaining cycles from 9000 to 6000 cycles. Essentially, the mode switching check-point can be thought of as a virtual entry point for the rest of the application.

As we take latency penalty into account, the functionality of the checkpoint only needs a small modification. In this case, instead of comparing the number of remaining cycles to $WCRC$ check-point would compare remaining cycles to $WCRC + S_d$ where S_d is the switching delay in cycles. If the number of remaining cycles is reasonably larger than $WCRC + S_d$, it means that we can scale down the frequency. If worst case remaining cycle is larger than $WCRC$ but not reasonably larger than $WCRC + S_d$, the mode switch will not be executed. However, we will not waste the existing slack because the execution slack is captured in the number of remaining cycles, and it can be utilized in upcoming check-points.

We can insert check-points on three type of edges. The first category are the forward branches (Figure 2.a). The reason is that branching might create changes in the number of remaining cycles. In addition, check-points can be inserted on the edges that immediately follow a loop body (Figure 2.b). The reason we insert checkpoint here is that it is likely to have slack right after the loop, because the actual number of iterations in the loop can be less than the iterations in the worst case. The third option is to insert the check-points in the first basic block of the loop. By adding check-point in the first basic block, we can exploit the slack for each single iteration of the loop. We set minimum distance between two check-points during check-points insertion. Therefore we do not introduce redundant switch penalty brought by check-points. If the next desired frequency is between two processor frequencies, we set the process to run at the faster frequency to ensure meeting the deadline constraint. However, calculation of the elapsed time will take this into account, and will ultimately incorporate it into T_L at upcoming check-points.

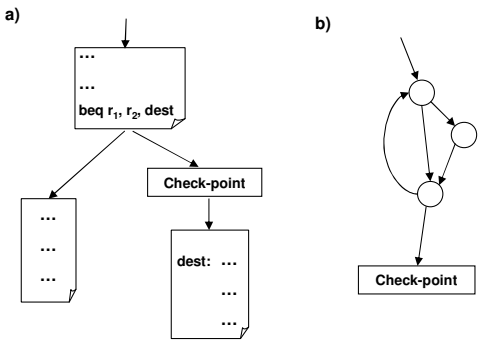

Figure 2. Check-points are inserted onto selected edges.

In order to determine *WCRC* values and insert check-points, our algorithm visits each edge of the application CFG only three times. Therefore, its runtime complexity is $O(m)$, where m is the number of edges in application control flow graph. For real applications, control flow graphs are sparse graphs in which, the number of control flow edges grows linearly with the number of nodes (basic blocks). Consequently, our algorithm runs very efficiently and is readily scalable to large applications.

6 Quantitative Analysis and Validation

6.1 Experimental Setup

In order to experiment the effectiveness of our technique, we have developed two compilation flows including well-known ILP-based [10] energy optimization and our proposed Check-Point Insertion Method (CPIM). Both of the compilers generate executable code for our target processor. Figure 3 illustrate our experimental setup for the CPIM.

We use MachineSUIF compiler framework [8] to extract the control flow graph of the applications. We simulate program performance and energy to estimate the power and latency of application basic blocks by using our XTREM-based cycle-accurate DVS+ABB simulator[10]. To perform the worst case analysis, we select the most complex input of each application as its *train input*. For each application, there are also a number of simpler input data would result in faster program execution. We refer to them as *test input*. By using the train input to profile the application, we can capture the WCET and the maximum number of loop iterations. After worst case analysis, we apply CPIM including *WCRC* analysis and checkpoint insertion. Finally we generate the code and simulate it using our simulation framework to measure the energy and performance of the generated code.

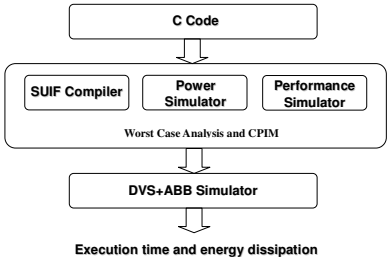

Figure 3. The setup of experiments for Check-Point Insertion Method (CPIM)

Table 2 summarizes the complexity, execution time, and compilation time for both ILP and CPIM for the selected applications from [7]. The selected application domains justify the need for execution deadline constraint and realtime operation of the generated code.

Table 2 reports the baseline execution time of the applications (using train input) with no frequency scaling when our proposed processor runs at $600MHz$. In order to investigate the effect of deadline relaxation on the quality of different frequency scaling methods, we have carried out extensive experiments using five different deadlines for each application. The first four deadlines are determined by averaging the adjacent execution times (e.g., execution time $@800MHz$ and $@600MHz$). For example, the first deadline is equal to the average execution time at $1GHz$ and $800MHz$ frequencies, with no frequency scaling mechanism. The last (fifth) deadline is set to 95% of the execution time at the slowest mode, i.e., running the processor at $200MHz$. Because of the page limitation, we can only report the experiment results for one of the deadlines in which, the deadline is the average of the execution time at $600MHz$ and $400MHz$. We would like to point out that the results are very consistent over different deadlines, and improvements generally grow with more relaxed deadlines.

6.2 Experimental Result

We implemented the experimental flows depicted in Figure 3 and generated code for five applications listed in Table 2. The compilation, corresponding simulations and analysis are performed using the train input. The train input is the one associated with worst case execution time (WCET).

The energy optimization techniques used in the experiment are ILP-based DVS only (without body bias), ILP-based DVS+ABB, and the CPIM-based DVS+ABB technique. To better measure the optimality gap of these techniques, we also adopted the analytical modeling and optimality analysis existing in the literature [4, 3], and applied it to our testbenches and processor model. When we calculate this analytical energy lower bound, we make some assumptions to the theoretical energy model of the processor. First of all, the ideal energy processor model has no switch latency and energy. Secondly, the ideal processor have exact the same frequency-voltage pairs as our realistic model and can not switch to other frequency or voltage arbitrarily. As for the memory, we keep it asynchronous with the processor in our ideal model. Then we use the simulator to determine the total execution cycles and time, total idle cycles and time for cache miss, and total cycles and time for processor operation.

According to [4], we need only two frequencies in the optimal discrete voltage schedule. We then apply the equation in [4] to calculate two consecutive frequencies for the analytical model. By using two consecutive frequencies and total time for processor operation, we can get the energy consumed by the processor operation. As for the energy consumed during the cache miss, we estimate the static energy consumption by applying the total cache miss time to the average static power consumption under zero bias voltage.

Note that the optimal energy dissipation predicted by such analytical modelings are only a lower bound on the amount of energy dissipation using any dynamic voltage scaling (either intra-task or inter-task) technique. The bounds are not tight, and in practice are not feasible.

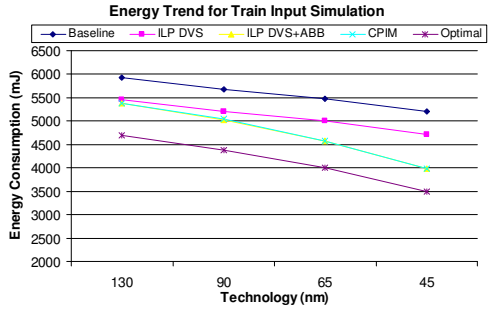

Figure 4. Energy Trend over the technology size for train input

Due to the page limitation, we are unable to report the experimental results for all of the hundred simulated cases (five applications under five deadlines and four technologies). We report the results only under one of the deadlines. Figure 4 and table 2 show that CPIM reduces the compilation time by more than three orders of magnitude for large programs, while achieving energy savings that are very close to the ILP-based results. More importantly, the results are only about $10-20\%$ away from the theoretical loose bound of energy savings, which means that our method comes reasonably close to the theoretical limit of the DVS+ABB technology.

According to our experiment result, CPIM acquires 14.3% energy saving for the baseline energy and outperform ILP based DVS about 6.16% in average under the 65nm technology because CPIM also optimize the leakage energy. Compared to the theoretical energy lower bound, CPIM is about 14.53% worse than the theoretical value. CPIM outperforms the ILP based DVS and DVS+ABB techniques by about 11.40% and 8.8% respectively in average under the 65nm technology. Compared to the theoretical energy lower bound, checkpoint method is about 19.65% away from the theoretical value. CPIM has the obvious advantage over the ILP-DVS technique for the advanced technology. The reason is that CPIM is designed for DVS+ABB optimization.

Figure 4 and 5 illustrates energy trend over the device sizes for our optimized techniques using train input and test input simulation respectively. The chart shows that the difference between baseline and optimized techniques becomes larger in the advanced technology. Therefore, the the energy saving will increase greatly as the device size shrinks.

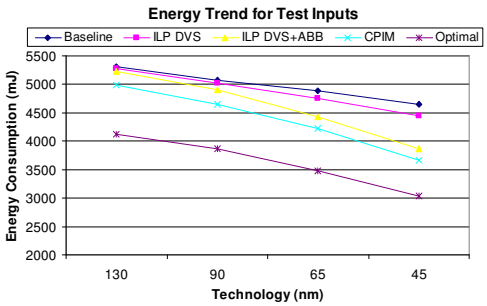

Figure 5. Energy Trend over the technology size for test inputs

Based on the experiment, CPIM can achieve the same energy saving but reduce the compilation time greatly. As for the compilation time, ILP technique is based on integer linear programming which grows exponentially with problem instance complexity. On the other hand, CPIM runtime depends on the number of basic blocks (control flow edges to be exact) in the application. Based on this observation, the complexity of CPIM will be in $O(m)$, where m means the total number of the edges in the application CFG. The

Benchmark	Application domain	# basic block	Exec. time @600MHz	Deadline	Average MILP solution time	Ave. Backtracking time	speed up
dijkstra	network	36	32.54	37.42	5.22	0.62	8.42
patricia	network	138	52.14	63.38	183	1.38	132.75
susan	automotive	203	43.14	52.8	1588	4.64	342.28
jpeg-dec	consumer	212	45.41	54.87	1613	4.63	348.49
gsm-dec	telecom	556	53.51	65.98	22451	14.69	1528.34

Table 2. Applications execution time and MILP solution time (sec)

CPIM visits all of the edges of the CFG three times, and runs in linear time of the input size.

CPIM even acquires more energy saving when we test our optimal setting by executing different inputs. Different from ILP technique, our heuristic will assign the operating mode of the processor based on the situation of the execution progress. If the execution progress goes to the non critical path, checkpoint can determine whether it is worth to execute a mode switch based on the existing slack. If the slack is not enough to outperform the switch latency, checkpoint will not execute the mode switch instruction. The checkpoint executes mode switch instruction only when it is worth to do that. Therefore, CPIM will not waste the slack in the meaningless mode switch instruction. If checkpoint does not execute the mode switch instruction, the executing overhead of the checkpoint is relative slight. However, ILP-based technique inserts mode switch instruction whose target operating mode is fixed to the edge of CDFG. Every fixed mode switch instruction will be executed regardless of the existence of the slack. Therefore, a difference on the control flow behavior between the train input and test input might downgrade the energy saving.

Since CPIM exploits the energy saving of the application by adaptively utilizing the existing slack, CPIM is closer to the theoretical limit of the energy saving when executing test inputs. However, ILP comes slightly closer to the theoretical limit of the energy saving when executing the train input. The reason is that ILP is the optimal solution for train input, while the static operating mode setting of the ILP method can not acquire optimal energy saving for different test inputs.

When we have some small slack can not be used to scale down the processor frequency, checkpoint will leave these small slack in the worst case remaining cycles and those slacks can usually be used in the future. This mechanism increase the opportunities to make use of the slack and reduce the waste of the slack when the number of the scaling frequency are few.

7 Conclusions

We present a methodology to combine dynamic voltage scaling and adaptive body biasing during compilation of an application targeting a DVS+ABB enabled embedded processor. Compiler-level analysis is particularly useful for embedded and realtime systems that demand light-weight operating systems. Additionally, compilers can exploit program execution trace information that are not visible to the operating system, and hence, can assist dynamic voltage schedulers. Experimental results show that the energy dissipation gap between leakage-aware and conventional DVS grows with technology scaling. Moreover, they show that our compiler's result come reasonably close to theoretical limits of energy savings using this method.

References

[1] D. Duarte, N. Vijaykrishnan, M. J. Irwin, H.-S. Kim, G. McFarland. "Impact of scaling on the effectiveness of dynamic power reduction schemes". In *Proceeding of International Conference on Computer Design*, pages 382–387, September 2002.

[2] D. Duarte, Y. Tsai, N. Vijaykrishnan, M. J. Irwin. "Evaluating runtime techniques for leakage power reduction". In *Proceeding of International Conference on VLSI Design*, pages 31–38, January 2002.

[3] F. Xie, M. Martonosi , S. Malik. "Intraprogram Dynamic Voltage Scaling:Bounding Opportunities with Analytic Modeling". *ACM Transactions on Architecture and Code Optimization*, 1(3):1–45, September 2004.

[4] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *International Conference on Computer Aided Design*, pages 560–563, 2001.

[5] http://www-device.eecs.berkeley.edu/ ptm/introduction.html.

[6] A. C. J.T. Kao, M. Miyazaki. "A 175-mv Multiply-Accumulate Unit Using an Adaptive Supply Voltage and Body Bias Architecture". *Journal of Solid-State Circuits*, 37(11):1545–1554, November 2002.

[7] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, and T. Mudge. "Mibench: a free, commercially representative embedded benchmark suite". In *Proceeding of the IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, December 2001.

[8] M.D. Smith, and G. Holloway. "An introduction to machine SUIF and its portable libraries for analysis and optimization". Technical report, Division of Engineering and Applied Sciences, Harvard University, 2002.

[9] N. AbouGhazaleh, D. Mossé, B.R. Childers, R. G. Melhem, M. Craven. Collaborative operating system and compiler power management for real-time applications. In *IEEE Real Time Technology and Applications Symposium*, pages 133–143, 2003.

[10] P. Huang, S. Ghiasi. "Leakage-Aware Intraprogram Voltage Scaling for Embedded Processors". In *Proc. Design Automation Conference*, pages 364–369, 2006.

[11] S.M. Martin, K. Flautner, T. Mudge, D. Blaauw. "Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads". In *Proceedings of the international conference on Computer-aided design*, pages 721–725, 2002.

[12] V. Suhendra, T. Mitra, A. Roychoudhury, T. Chen. "Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis". In *Proc. Design Automation Conference*, pages 358–363,2006.